Contents

1 Preamble **2**

2 Signature **3**

3 Documentation **5**

3.1 Constructing a Matrix 6
3.2 Deconstructing a Matrix 7
3.3 Combinators and Higher-Order Functions 9
3.4 Indexing-Related Functions 10

1 Preamble

The `MATRIX` signature encodes a specification for two-dimensional matrices. Many of the functions are analogous to those in the `SEQUENCE` signature.

Indices are defined as `type index = int * int`, representing the row and column, respectively.

2 Signature

```
signature MATRIX =
sig

  type 'a t
  type 'a mat = 'a t

  (* row, column *)
  type index = int * int

  exception Range

  (* Constructing a Matrix *)

  val tabulate : (index -> 'a) -> index -> 'a mat

  val fromRows : 'a Seq.t Seq.t -> 'a mat
  val fromCols : 'a Seq.t Seq.t -> 'a mat


  (* Deconstructing a Matrix *)

  val nth : 'a mat -> index -> 'a
  val size : 'a mat -> index
  val toSeq : 'a mat -> 'a Seq.t
  val toString : ('a -> string) -> 'a mat -> string

  val rows : 'a mat -> 'a Seq.t Seq.t
  val cols : 'a mat -> 'a Seq.t Seq.t

  val neighborsIdx : 'a mat -> index -> index Seq.t
  val neighbors : 'a mat -> index -> 'a Seq.t


  (* Combinators and Higher-Order Functions *)

  val map : ('a -> 'b) -> 'a mat -> 'b mat
  val reduce : ('a * 'a -> 'a) -> 'a -> 'a mat -> 'a
  val zip : ('a mat * 'b mat) -> ('a * 'b) mat
  val zipWith : ('a * 'b -> 'c) -> 'a mat * 'b mat -> 'c mat


  (* Indexing-Related Functions *)

  val enum : 'a mat -> (index * 'a) mat
  val mapIdx : (index * 'a -> 'b) -> 'a mat -> 'b mat
  val update : 'a mat -> (index * 'a) -> 'a mat
  val inject : 'a mat -> (index * 'a) Seq.t -> 'a mat

```

```
end
```

3 Documentation

We assume that all functions that are given as arguments (such as the `f` in `map f`) have $O(1)$ work and span. In order to analyze the runtime of matrix functions when this is not the case, we need to analyze the corresponding cost graphs.

Given a matrix `M` with `size M` $\cong$ `(m,n)`, we define the notation $|M| =$ `m` $\cdot$ `n`.

Definition 1 (Associative)**.** Fix some type `t`. We say a function `g : t * t -> t` is *associative* if for all `a`, `b`, and `c` of type `t`:

`g (g (a,b),c)` $\cong$ `g (a,g (b,c))`

Definition 2 (Commutative)**.** Fix some type `t`. We say a function `g : t * t -> t` is *commutative* if for all `a` and `b` of type `t`:

`g (a,b)` $\cong$ `g (b,a)`

Definition 3 (Identity)**.** Fix some type `t`. Given a function `g : t * t -> t`, we say `z` is the *identity* for `g` if for all `x : t`:

`g (x,z)` $\cong$ `g (z,x)` $\cong$ `x`

3.1 Constructing a Matrix

`tabulate : (index -> 'a) -> index -> 'a mat`

REQUIRES: For all $0 \leq$ `i` $<$ `m` and $0 \leq$ `j` $<$ `n`, `f (i,j)` is valuable.

ENSURES: `tabulate f (m,n)` $\Longrightarrow$ `M` where `size M` $\cong$ `(m,n)` and for all $0 \leq$ `i` $<$ `m` and $0 \leq$ `j` $<$ `n`, `nth M (i,j)` $\cong$ `f (i,j)`.

Raises `Range` if `m` or `n` are negative.

Work $O(\mathtt{m} \cdot \mathtt{n})$, Span $O(1)$.

`fromRows : 'a Seq.t Seq.t -> 'a mat`

REQUIRES: `s` is non-empty with length `m` and all elements of `s` have the same length `n`.

ENSURES: `fromRows s` $\Longrightarrow$ `M` where

`nth M (i,j)` $\cong$ `Seq.nth (Seq.nth s i) j`

Work $O(\mathtt{m} \cdot \mathtt{n})$, Span $O(1)$.

`fromCols : 'a Seq.t Seq.t -> 'a mat`

REQUIRES: `s` is non-empty with length `n` and all elements of `s` have the same length `m`.

ENSURES: `fromCols s` $\Longrightarrow$ `M` where

`nth M (i,j)` $\cong$ `Seq.nth (Seq.nth s j) i`

Work $O(\mathtt{m} \cdot \mathtt{n})$, Span $O(1)$.

3.2 Deconstructing a Matrix

```
nth : 'a mat -> index -> 'a
```

ENSURES: `nth M (i,j)` evaluates to the `(i,j)`$^{\text{th}}$ element of `M`. Raises `Range` if either $0 \le$ `i` $<$ `m` or $0 \le$ `j` $<$ `n` fail to be met, given that `size M` $\cong$ `(m,n)`.

Work $O(1)$, Span $O(1)$.

```
size : 'a mat -> index
```

ENSURES: `size M` evaluates to `(m,n)`, representing the rows and columns of `M`, respectively.

Work $O(1)$, Span $O(1)$.

```
toSeq : 'a mat -> 'a Seq.t
```

ENSURES: `toSeq M` evaluates to a sequence consisting of the elements in `M`.

Work $O(1)$, Span $O(1)$.

```
toString : ('a -> string) -> 'a mat -> string
```

REQUIRES: `ts x` evaluates to a value for all elements `x` in `M` (e.g. if `ts` is total).

ENSURES: `toString ts M` evaluates to a string representation of `M`, using the function `ts` to convert each element of `M` into a string.

Work $O(|\texttt{M}|)$, Span $O(\log |\texttt{M}|)$.

```
rows : 'a mat -> 'a Seq.t Seq.t
```

ENSURES: `rows M` evaluates to a sequence of length `m`, with each element having length `n` and representing the rows of `M` in order, given `size M` $\cong$ `(m,n)`.

Work $O(\texttt{m})$, Span $O(1)$.

```
cols : 'a mat -> 'a Seq.t Seq.t
```

ENSURES: `cols M` evaluates to a sequence of length `n`, with each element having length `m` and representing the columns of `M` in order, given `size M` $\cong$ `(m,n)`.

Work $O(\texttt{m} \cdot \texttt{n})$, Span $O(1)$.

```
neighborsIdx : 'a mat -> index -> index Seq.t
```

REQUIRES: $0 \le$ `i` $<$ `m` and $0 \le$ `j` $<$ `n`, given `size M` $\cong$ `(m,n)`

ENSURES: `neighborsIdx M (i,j)` evaluates to a sequence containing the indices valid in `M` which are adjacent to `(i,j)`, including horizontally, vertically, and diagonally.

Work $O(1)$, Span $O(1)$.

`neighbors : 'a mat -> index -> 'a Seq.t`

REQUIRES: $0 \leq$ `i` $<$ `m` and $0 \leq$ `j` $<$ `n`, given `size M` $\cong$ `(m,n)`

ENSURES: `neighbors M (i,j)` evaluates to a sequence containing the elements of `M` which are adjacent to index `(i,j)`, including horizontally, vertically, and diagonally.

Work $O(1)$, Span $O(1)$.

3.3 Combinators and Higher-Order Functions

```
map : ('a -> 'b) -> 'a mat -> 'b mat
```

REQUIRES: `f x` evaluates to a value for all elements `x` of `M` (e.g. if `f` is total).

ENSURES: `map f M` $\Longrightarrow$ `M'` such that $|\texttt{M}| = |\texttt{M'}|$ and for all $0 \leq \texttt{i} < \texttt{m}$ and $0 \leq \texttt{j} < \texttt{n}$, `nth M' (i,j)` $\cong$ `f (nth M (i,j))`, given `size M` $\cong$ `(m,n)`.

Work $O(|\texttt{M}|)$, Span $O(1)$.

```
reduce : ('a * 'a -> 'a) -> 'a -> 'a mat -> 'a
```

REQUIRES:

- `g` is total, associative, and commutative.
- `z` is the identity for `g`.

ENSURES: `reduce g z M` uses the function `g` to combine the elements of `M` using `z` as a base case.

Work $O(|\texttt{M}|)$, Span $O(\log |\texttt{M}|)$.

```
zip : 'a mat * 'b mat -> ('a * 'b) mat
```

REQUIRES: $|\texttt{M1}| = |\texttt{M2}|$

ENSURES: `zip (M1,M2)` $\Longrightarrow$ `M'` such that $|\texttt{M}| = |\texttt{M1}| = |\texttt{M2}|$ such that for all $0 \leq \texttt{i} < \texttt{m}$ and $0 \leq \texttt{j} < \texttt{n}$, `nth M (i,j)` $\cong$ `(nth M1 (i,j),nth M2 (i,j)`, given `size M1` $\cong$ `(m,n)`.

Work $O(|\texttt{M1}|)$, Span $O(1)$.

```
zipWith : ('a * 'b -> 'c) -> 'a mat * 'b mat -> 'c mat
```

REQUIRES: $|\texttt{M1}| = |\texttt{M2}|$

ENSURES: `zipWith f (M1,M2)` $\cong$ `map f (zip (M1,M2))`.

Work $O(|\texttt{M1}|)$, Span $O(1)$.

3.4 Indexing-Related Functions

`enum : 'a mat -> (index * 'a) mat`

ENSURES: `enum M` $\Longrightarrow$ `M'` such that for all $0 \le$ `i` $<$ `m` and $0 \le$ `j` $<$ `n`, given `size M` $\cong$ `(m,n)`, `nth M' (i,j)` $\cong$ `((i,j),nth M (i,j))`.

Work $O(|\texttt{M}|)$, Span $O(1)$.

`mapIdx : (index * 'a -> 'b) -> 'a mat -> 'b mat`

ENSURES: `mapIdx f M` $\cong$ `map f (enum M)`.

Work $O(|\texttt{M}|)$, Span $O(1)$.

`update : 'a mat * (index * 'a) -> 'a mat`

ENSURES: `update (M,((i,j),x))` evaluates to a matrix identical to `M` but with the `(i,j)`$^{\text{th}}$ element now `x` if $0 \le$ `i` $<$ `m` and $0 \le$ `j` $<$ `n`, given `size M` $\cong$ `(m,n)`, and raises `Range` otherwise.

Work $O(|\texttt{M}|)$, Span $O(1)$.

`inject : 'a mat * (int * 'a) Seq.t -> 'a mat`

ENSURES: `inject (M,U)` evaluates to a matrix where for each `((i,j),x)` in `U`, the `(i,j)`$^{\text{th}}$ element of `M` is replaced with `x`. If there are multiple elements at the same index, one is chosen nondeterministically. If any indices are out of bounds, raises `Range`.

Work $O(|\texttt{M}| + |\texttt{U}|)$, Span $O(1)$.